Dependabilidade

Serviço, Falhas e Tolerância

Prof. Dr. Iaçanã Ianiski Weber

Confiabilidade e Segurança de Software

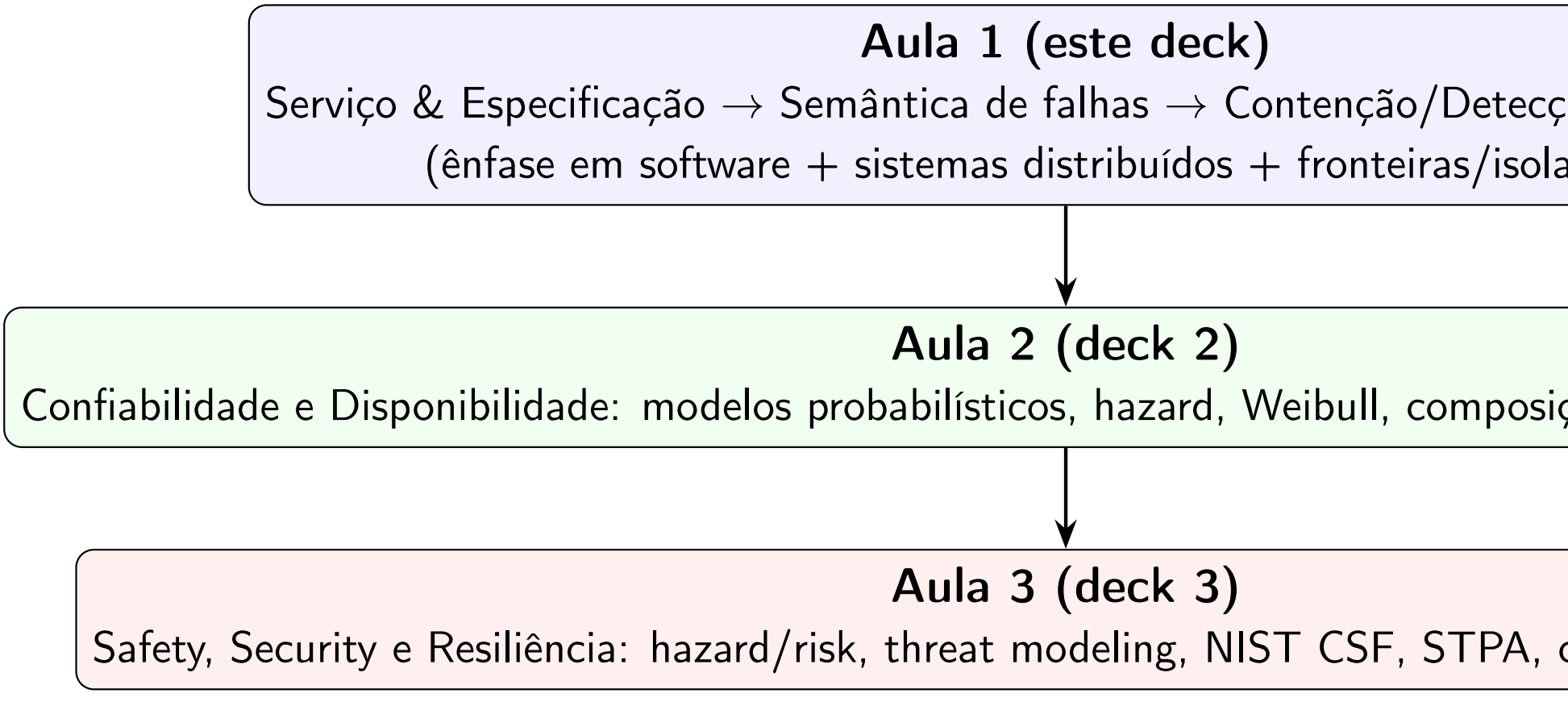
98G08-4

Escopo desta aula: "antes" de métricas (Aula 2) e "antes" de safety/security/resiliência (Aula 3).

Índice

Índice

Mapa das 3 aulas (para evitar sobreposição)



Mensagem

Se você não define **qual é o serviço** e **qual semântica de falha** você tolera, as métricas (Aula 2) e os requisitos de safety/security (Aula 3) ficam "soltos".

Objetivos técnicos da Aula 1 (revisados)

Ao final, o aluno deve conseguir:

- Definir **serviço** e **especificação** e entender por que **falha** é desvio do contrato (não "bug").
- Usar a cadeia *fault/defeito*$\rightarrow$ *erro de estado*$\rightarrow$ *falha*, mas com foco em:
  - **semântica de falhas** (crash/omission/timing/value/Byzantine),
  - **semântica de execução** em RPC (at-least-once / at-most-once) e suas implicações.
- Projetar **barreiras**: detecção, contenção (fault containment) e recuperação (restart/rollback/transação).
- Identificar armadilhas clássicas: retries amplificando incidentes, idempotência, causa comum, observabilidade insuficiente.

Índice

Serviço é o que o mundo observa (e cobra)

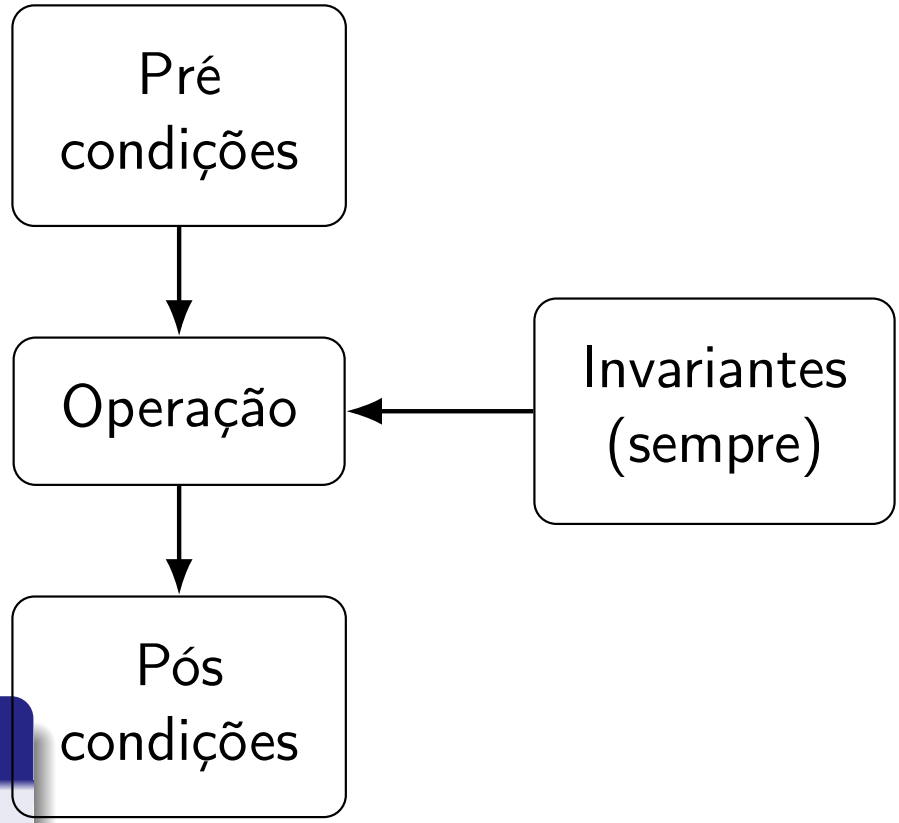
Definição operacional

Serviço (*serviço*) é o comportamento observável na interface com usuário/sistemas;
Estado interno pode estar errado (*erro de estado*) e mesmo assim o serviço não falhar (mas isso é risco!).

- A mesma implementação pode "falhar" ou "não falhar" dependendo da *especificação*:
  - latência máxima (timing), valor correto (value), disponibilidade, consistência.
- Em sistemas modernos, "falha" pode ser **parcial**:
  - degradação (p99 explode), funcionalidade limitada, respostas inconsistentes.

Contrato mínimo: pré/pós-condições e invariantes

- **Pré-condições**: o que precisa ser verdade antes da operação (inputs válidos, permissões).
- **Pós-condições**: o que deve ser verdade após (efeitos e retornos).
- **Invariantes**: propriedades que não podem ser violadas (ex.: saldo nunca negativo).



Por que isso é dependabilidade?

Porque dá **alvos explícitos** para detecção de *erro de estado*e contenção antes da *falha*.

Fault, Error, Failure (1 slide e seguimos adiante)

Fault causa potencial (bug, requisito incompleto, config errada, desgaste, operador).

Error parte do estado incorreto (ex.: contador corrompido, ponteiro inválido, cache inconsistente).

Failure desvio do serviço frente à *especificação*(ex.: valor errado, timeout, indisponibilidade).

Ponto-chave desta aula

O projeto real gira em torno de: **onde** o error pode surgir, **como** detectar, **como** conter, **como** recuperar.

Índice

Falha não é única: classes que mudam a arquitetura

Classe	**Exemplo de desvio de serviço**
Crash	componente para de responder
Omission	mensagens/ações "somem" (perda)
Timing	resposta fora da janela (deadline estourado)
Value/Response	responde com valor incorreto
Byzantine	comportamento arbitrário/inconsistente

Regra de ouro

Você não "faz tolerância a falhas" em abstrato: você tolera **um modelo de falha declarado**.

Fail-silent, fail-stop, fail-fast (o vocabulário certo)

- **Fail-silent**: componente falha ficando "mudo" (sem sinais/saídas erradas).
- **Fail-stop**: além de parar, outros componentes conseguem **detectar** que ele parou.
- **Fail-fast**: ao detectar inconsistência, aborta rapidamente para evitar propagação de *erro de estado*.

Conexão com projeto

Assumir fail-stop quando o sistema é na prática value-failure é uma das raízes de incidentes difíceis (falha "falando coisa errada").

Índice

RPC: onde a intuição do aluno quebra

Problema

Em RPC (rede + processos), não dá para observar diretamente se o servidor executou a operação quando há falhas de comunicação.

Após um timeout, pode ter acontecido:

- não executou,
- executou 1 vez (mas a resposta se perdeu),
- executou múltiplas vezes (retries),
- executou parcialmente (efeitos colaterais).

Semântica de execução: at-least-once vs at-most-once

at-least-once

- Cliente reenvia até obter resposta.
- Pode **executar múltiplas vezes**.
- Exige: **idempotência** ou **deduplicação** para operações com efeitos colaterais.

at-most-once

- Evita múltiplas execuções via **IDs + cache de respostas / tabelas de request**.
- Pode falhar por **omissão** (não executar) sob certas perdas.
- Requer **estado** e **coleta** (GC) desses registros.

Mensagem

Tabela mental: por que o cliente não sabe o que ocorreu

Evento de falha	**Sintoma**	**Efeito possível**
Request perdido	timeout	servidor **não** executa
Response perdida	timeout	servidor executa, cliente **não sabe**
Servidor crash após executar	timeout/erro	efeitos ocorreram, mas estado pode estar incerto
Partição de rede	timeouts	retries podem amplificar carga (incidente em cascata)

Conseqüência

Retries são mecanismo de tolerância **e** uma fonte comum de amplificação de falhas se não houver backoff/jitter e controle de explosão.

Idempotência: a alavanca mais importante (exemplo simples)

```
# Operacao NAO idempotente: "debitar" duas vezes = prejuizo
def debit(account_id, amount):
    balance[account_id] -= amount

# Operacao idempotente por chave (idempotency key):
# repetir a requisicao nao muda o efeito final.
def debit_idempotent(account_id, amount, req_id):
    if req_id in processed: # deduplicacao
        return processed[req_id] # replay da resposta
    balance[account_id] -= amount
    processed[req_id] = "OK"
    return "OK"
```

Índice

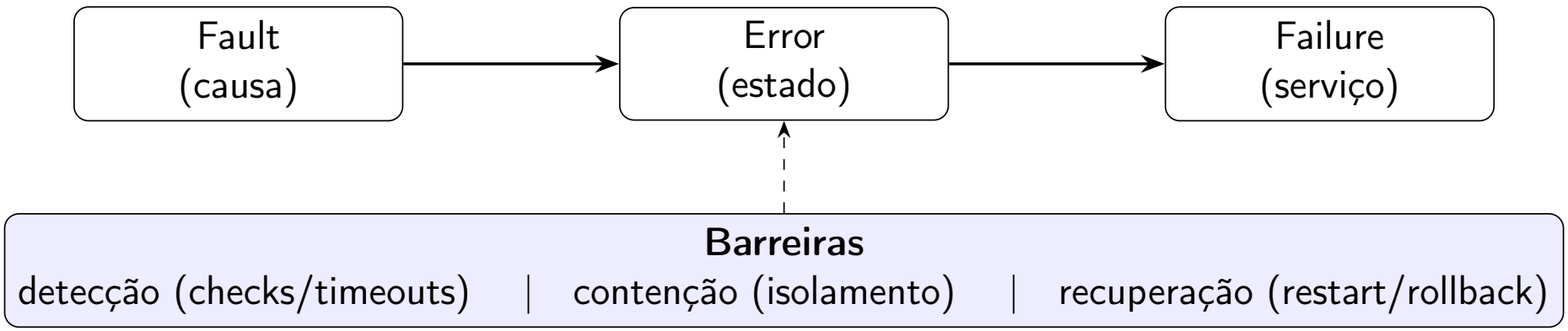
Fault Containment: reduzir o "blast radius"

Ideia

Uma arquitetura tolerante a falhas delimita **regiões de contenção**: onde um *erro de estado* pode existir sem virar *falha* sistêmica.

- Contenção é tão importante quanto recuperação:
  - impede corrupção de estado global,
  - evita cascatas (um módulo derrubando outros),
  - facilita diagnóstico (localiza causa).
- Em software: contenção aparece como **fronteiras**:
  - processo, container, VM, domínio de privilégio, partição de barramento, sandbox.

Barreiras clássicas: detectar → conter → recuperar



End-to-end argument (por que checks na borda importam)

Princípio (intuição)

Algumas propriedades só podem ser garantidas **de ponta a ponta** na aplicação (ex.: integridade real do dado), então checks internos (rede/armazenamento) ajudam, mas **não substituem** o check end-to-end.

- Exemplo: checksum em cada hop vs checksum final no payload.
- Em dependabilidade: isso define **onde** colocar detecção para impedir *erro de estado*$\rightarrow$*falha*.

Índice

Detecção prática: checks que pegam erros cedo

- **Checks semânticos**: range checks, sanity checks, contratos, invariantes.
- **Checks estruturais**: CRC/checksum, paridade, ECC (onde aplicável).
- **Checks temporais**: timeouts, deadlines, heartbeats.
- **Checks por redundância**: dual execution, comparação, votação (quando necessário).

Escolha de engenharia

Checks bons são aqueles que:

- detectam cedo,
- têm baixa taxa de falso positivo,
- e são verificáveis/testáveis.

Diagnosabilidade: sem telemetria, MTTR explode (sem entrar em métricas)

- Logs estruturados e correlacionáveis (ex.: **request-id**).
- Tracing distribuído (quando há microserviços).
- "Black box vs white box": o que você mede precisa apoiar hipótese causal.
- Preserve evidência: dumps, snapshots, ring buffers, post-mortem.

Mensagem

Dependabilidade em software não é só "rodar sem falhar": é **falhar de forma diagnosticável**.

Índice

Recuperação: quatro estratégias (com trade-offs)

1. **Restart** (reboot/restart de processo): simples, mas pode perder estado.
2. **Rollback** (checkpoint + voltar): requer pontos consistentes.
3. **Transação** (atomicidade/durabilidade): evita estados parciais, mas custa.
4. **Compensação** (sagas): desfaz efeitos em sistemas distribuídos quando transação global é inviável.

Ponto crítico

A estratégia depende de: **(i)** semântica do serviço, **(ii)** custo de estado inconsistente, **(iii)** latência aceitável.

Transações: o que você quer evitar (estado parcial)

- Problema: falha no meio de uma sequência deixa o sistema em estado "meio aplicado".
- Abordagens:
  - **Write-ahead logging** (log durável antes do commit),
  - **two-phase commit** (quando aplicável; caro e frágil sob partições),
  - **sagas/compensações** em arquiteturas modernas.

Recuperação e contenção andam juntas (senão vira cascata)

- Se um serviço A falha e B faz retry agressivo:
  - B aumenta carga em A,
  - A degrada mais,
  - A derruba dependências (cascata).
- Mitigações típicas:
  - backoff/jitter, rate limiting, circuit breaker, filas.

Índice

Caso 1 (software/ops): Knight Capital (2012) como falha de mudança + contenção

- Contexto: mudança em software de trading; ativação de código/feature em produção.
- Efeitos: emissão de ordens inesperadas em alta escala em minutos.
- Pontos de engenharia (dependability):
  - controle de rollout/configuração,
  - isolamento e limites de dano (circuitos, rate limits),
  - observabilidade e "kill switch".

Caso 2 (integração e suposições): Ariane 5 (1996) como falha de envelope

- Reuso de software + hipótese inválida fora do envelope operacional.
- Uma exceção em software se propagou para perda de orientação e perda do veículo.

Lição focada (Aula 1)

Semântica de falha + contenção: quando um subsistema crítico falha, ele deve falhar de forma **contida** (não virar "saída plausível porém errada" para o controlador).

Índice

Exercícios (1/3): semântica de falhas e especificação

1. Uma API de pagamento tem *especificação*: "debitar no máximo 1 vez por pedido".
   - Que semântica de execução você precisa?
   - Qual o mecanismo mínimo (idempotency key / dedup / transação)?
2. Um sistema de controle tem deadline de 10ms.
   - Timing failure é *falha*? Por quê?
   - Proponha 2 checks (temporal e semântico).

Exercícios (2/3): contenção e recuperação

1. Identifique 3 fronteiras de contenção possíveis em um sistema IoT: (processo, container, MCU+MPU, gateway, cloud). Para cada uma, diga: **o que ela contém** e **o que não contém**.
2. Proponha uma estratégia de recuperação para:
   - (a) deadlock de firmware (restart),
   - (b) corrupção de estado (rollback/checkpoint),
   - (c) efeitos distribuídos (compensação).

Exercícios (3/3): armadilhas clássicas

1. Por que retries podem derrubar um sistema já degradado?
2. O que significa "cobertura de detecção" (coverage) e por que importa?
3. Dê um exemplo de "causa comum" que derrota redundância em software (mesma config/mesmo bug).

Resumo da Aula 1 (sem sobrepor com Aula 2/3)

- Falha é desvio do **serviço** frente à **especificação**.
- Semântica de falhas e de execução (RPC) define escolhas: idempotência, dedup, timeouts e retries.
- Arquitetura tolerante a falhas = **detectar** cedo, **conter** dano, **recuperar** com estratégia apropriada.
- Projetar para **diagnosticar** é parte da dependabilidade (sem isso, o reparo vira "arte").

Gancho

Na Aula 2, vamos quantificar (modelos/métricas) e compor sistemas; depois, na Aula 3, ampliamos para safety/security/resiliência.

Referências abertas (seleção para este deck) I

- Avizienis, Laprie, Randell, Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE TDSC, 2004. https://www.landwehr.org/2004-aviz-laprie-randell.pdf
- Stanford (CS244b / Distributed Systems). Notas sobre falhas, RPC e semântica de execução (materiais abertos variam por edição).
- CMU (15-440 / Distributed Systems). Materiais de RPC, timeouts e semânticas (at-least/at-most once).
- MIT (6.033 / Computer Systems Engineering). Leituras/lectures sobre RPC, falhas parciais e trade-offs de semântica.
- JPL/NASA (fault management). Linguagem útil sobre contenção/isolamento e estratégias de fault management.
- Knight Capital (2012). Documentos regulatórios/relatórios (SEC) sobre controles e falhas de tecnologia.